Java常见的几种排序算法-插入、选择、冒泡、快排、堆排等

本文就是介绍一些常见的排序算法。排序是一个非常常见的应用场景，很多时候，我们需要根据自己需要排序的数据类型，来自定义排序算法，但是，在这里，我们只介绍这些基础排序算法，包括：插入排序、选择排序、冒泡排序、快速排序（重点）、堆排序、归并排序等等。看下图：



给定数组：int data[] = {9,2,7,19,100,97,63,208,55,78}

一、直接插入排序(内部排序、O(n2)、稳定)

原理：从待排序的数中选出一个来，插入到前面的合适位置。

package com.xtfggef.algo.sort;

public class InsertSort {

static int data[] = { 9, 2, 7, 19, 100, 97, 63, 208, 55, 78 };

public static void insertSort() {

int tmp, j = 0;

for (int k = 0; k < data.length; k++) {//-----1-----

tmp = data[k];

j = k - 1;

while (j >= 0 && tmp < data[j]) {//-----2-----

data[j + 1] = data[j];

j--;

}

data[j + 1] = tmp;//------3-------

}

}

public static void main(String[] args) {

print();

System.out.println();

insertSort();

System.out.println();

print();

}

static void print() {

for (int i = 0; i < data.length; i++) {

System.out.print(data[i] + " ");

}

}

}

我简单的讲解一下过程：思路上从待排序的数据中选出一个，插入到前面合适的位置，耗时点在插入方面，合适的位置意味着我们需要进行比较找出哪是合适的位置，举个例子：对于9,2,7,19,100,97,63,208,55,78这组数，第一个数9前面没有，不做操作，当第一个数完后，剩下的数就是待排序的数，我们将要从除去9开始的书中选出一个插入到前面合适的位置，拿到2后，放在tmp上，进行注释中的2处的代码，2处的代码就是通过循环找出这个合适的位置，发现比tmp大的数，立即将该数向后移动一位（这样做的目的是：前面需要空出一位来进行插入），最后通过注释3处的代码将数插入。

本排序适合：基本有序的数据

二、选择排序（O(n2)、不稳定）

与直接插入排序正好相反，选择排序是从待排序的数中选出最小的放在已经排好的后面，这个算法选数耗时。

package com.xtfggef.algo.sort;

public class SelectSort {

static int data[] = { 9, 2, 7, 19, 100, 97, 63, 208, 55, 78 };

public static void selectSort() {

int i, j, k, tmp = 0;

for (i = 0; i < data.length - 1; i++) {

k = i;

for (j = i + 1; j < data.length; j++)

if (data[j] < data[k])

k = j;

if (k != i) {

tmp = data[i];

data[i] = data[k];

data[k] = tmp;

}

}

}

public static void main(String[] args) {

print();

System.out.println();

selectSort();

System.out.println();

print();

}

static void print() {

for (int i = 0; i < data.length; i++) {

System.out.print(data[i] + " ");

}

}

}

通过循环，找出最小的数的下标，赋值于k，即k永远保持待排序数据中最小的数的下标，最后和当前位置i互换数据即可。

三、快速排序（O(nlogn)、不稳定）

快速排序简称快排，是一种比较快的排序，适合基本无序的数据，为什么这么说呢？下面我说下快排的思路：

设置两个指针：i和j，分别指向第一个和最后一个，i像后移动，j向前移动，选第一个数为标准（一般这样做，当然快排的关键就是这个“标准”的选取），从后面开始，找到第一个比标准小的数，互换位置，然后再从前面，找到第一个比标准大的数，互换位置，第一趟的结果就是标准左边的都小于标准，右边的都大于标准（但不一定有序），分成两拨后，继续递归的使用上述方法，最终有序！代码如下：

package com.xtfggef.algo.sort;

public class QuickSortTest {

static class QuickSort {

public int data[];

private int partition(int array[], int low, int high) {

int key = array[low];

while (low < high) {

while (low < high && array[high] >= key)

high--;

array[low] = array[high];

while (low < high && array[low] <= key)

low++;

array[high] = array[low];

}

array[low] = key;

return low;

}

public int[] sort(int low, int high) {

if (low < high) {

int result = partition(data, low, high);

sort(low, result - 1);

sort(result + 1, high);

}

return data;

}

}

static void print(int data[]) {

for (int i = 0; i < data.length; i++) {

System.out.print(data[i] + " ");

}

}

public static void main(String[] args) {

int data[] = { 20, 3, 10, 9, 186, 99, 200, 96, 3000 };

print(data);

System.out.println();

QuickSort qs = new QuickSort();

qs.data = data;

qs.sort(0, data.length - 1);

print(data);

}

}

看看上面的图，基本就明白了。

四、冒泡排序（稳定、基本有序可达O(n)，最坏情况为O(n2)）

冒泡排序是一种很简单，不论是理解还是时间起来都比较容易的一种排序算法，思路简单：小的数一点一点向前起泡，最终有序。

package com.xtfggef.algo.sort;

public class BubbleSort {

static int data[] = { 9, 2, 7, 19, 100, 97, 63, 208, 55, 78 };

public static void bubbleSort() {

int i, j, tmp = 0;

for (i = 0; i < data.length - 1; i++) {

for (j = data.length - 1; j > i; j--) {

if (data[j] < data[j - 1]) {

tmp = data[j];

data[j] = data[j - 1];

data[j - 1] = tmp;

}

}

}

}

public static void main(String[] args) {

print();

System.out.println();

bubbleSort();

System.out.println();

print();

}

static void print() {

for (int i = 0; i < data.length; i++) {

System.out.print(data[i] + " ");

}

}

}

五、堆排序

我们这里不详细介绍概念，堆的话，大家只要记得堆是一个完全二叉树（什么是完全二叉树，请不懂的读者去查资料），堆排序分为两种堆，大顶堆和小顶堆，大顶堆的意思就是堆顶元素是整个堆中最大的，小顶堆的意思就是堆顶元素是整个堆中最小的，满足：任何一非叶节点的关键字不大于或者不小于其左右孩子节点的关键字。堆排序是一个相对难理解的过程，下面我会较为清楚、详细的讲解一下堆排序。堆排序分为三个过程：

建堆：从一个数组顺序读取元素，建立一个堆（完全二叉树）

初始化：将堆进行调整，使得堆顶为最大（最大堆）或者最小（最小堆）的元素

维护：将堆顶元素出堆后，需要将堆的最后一个节点补充到堆顶，因为这样破坏了堆的秩序，所以需要进行维护。下面我们图示一下：

一般情况，建堆和初始化同步进行，

最后为如下所示，即为建堆、初始化成功。

我们可以观察下这个最大堆，看出堆顶是整个堆中最大的元素，而且除叶子节点外每个节点都大于其子节点。下面的过程就是当我们输出堆顶元素后，对堆进行维护。

过程是这样：将堆顶元素出堆后，用最后一个元素补充堆顶元素，这样破坏了之前的秩序，需要重新维护堆，在堆顶元素的左右节点中选出较小的和堆顶互换，然后一直递归下去，所以每次出一个元素，需要一次维护，堆排序适合解决topK问题，能将复杂度降到nlogK。下面是代码：

package com.xtfggef.algo.sort;

public class HeapSort {

private static int[] sort = new int[] { 1, 0, 10, 20, 3, 5, 6, 4, 9, 8, 12,

17, 34, 11 };

public static void main(String[] args) {

buildMaxHeapify(sort);

heapSort(sort);

print(sort);

}

private static void buildMaxHeapify(int[] data) {

// 没有子节点的才需要创建最大堆，从最后一个的父节点开始

int startIndex = getParentIndex(data.length - 1);

// 从尾端开始创建最大堆，每次都是正确的堆

for (int i = startIndex; i >= 0; i--) {

maxHeapify(data, data.length, i);

}

}

/\*\*

\* 创建最大堆

\*

\* @param data

\* @param heapSize

\* 需要创建最大堆的大小，一般在sort的时候用到，因为最多值放在末尾，末尾就不再归入最大堆了

\* @param index

\* 当前需要创建最大堆的位置

\*/

private static void maxHeapify(int[] data, int heapSize, int index) {

// 当前点与左右子节点比较

int left = getChildLeftIndex(index);

int right = getChildRightIndex(index);

int largest = index;

if (left < heapSize && data[index] < data[left]) {

largest = left;

}

if (right < heapSize && data[largest] < data[right]) {

largest = right;

}

// 得到最大值后可能需要交换，如果交换了，其子节点可能就不是最大堆了，需要重新调整

if (largest != index) {

int temp = data[index];

data[index] = data[largest];

data[largest] = temp;

maxHeapify(data, heapSize, largest);

}

}

/\*\*

\* 排序，最大值放在末尾，data虽然是最大堆，在排序后就成了递增的

\*

\* @param data

\*/

private static void heapSort(int[] data) {

// 末尾与头交换，交换后调整最大堆

for (int i = data.length - 1; i > 0; i--) {

int temp = data[0];

data[0] = data[i];

data[i] = temp;

maxHeapify(data, i, 0);

}

}

/\*\*

\* 父节点位置

\*

\* @param current

\* @return

\*/

private static int getParentIndex(int current) {

return (current - 1) >> 1;

}

/\*\*

\* 左子节点position 注意括号，加法优先级更高

\*

\* @param current

\* @return

\*/

private static int getChildLeftIndex(int current) {

return (current << 1) + 1;

}

/\*\*

\* 右子节点position

\*

\* @param current

\* @return

\*/

private static int getChildRightIndex(int current) {

return (current << 1) + 2;

}

private static void print(int[] data) {

int pre = -2;

for (int i = 0; i < data.length; i++) {

if (pre < (int) getLog(i + 1)) {

pre = (int) getLog(i + 1);

System.out.println();

}

System.out.print(data[i] + " |");

}

}

/\*\*

\* 以2为底的对数

\*

\* @param param

\* @return

\*/

private static double getLog(double param) {

return Math.log(param) / Math.log(2);

}

}

慢慢理解一下，还是容易明白的！

六、归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

首先考虑下如何将将二个有序数列合并。这个非常简单，只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

package com.xtfggef.algo.sort;

public class SortTest {

// 将有序数组a[]和b[]合并到c[]中

static void MemeryArray(int a[], int n, int b[], int m, int c[]) {

int i, j, k;

i = j = k = 0;

while (i < n && j < m) {

if (a[i] < b[j])

c[k++] = a[i++];

else

c[k++] = b[j++];

}

while (i < n)

c[k++] = a[i++];

while (j < m)

c[k++] = b[j++];

}

public static void main(String[] args) {

int a[] = { 2, 7, 8, 10, 299 };

int b[] = { 5, 9, 14, 20, 66, 88, 92 };

int c[] = new int[a.length + b.length];

MemeryArray(a, 5, b, 7, c);

print(c);

}

private static void print(int[] c) {

for (int i = 0; i < c.length; i++) {

System.out.print(c[i] + " ");

}

}

}

可以看出合并有序数列的效率是比较高的，可以达到O(n)。解决了上面的合并有序数列问题，再来看归并排序，其的基本思路就是将数组分成二组A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序。如何让这二组组内数据有序了？可以将A，B组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。这样通过先递归的分解数列，再合并数列就完成了归并排序。下面是归并排序代码：

package com.xtfggef.algo.sort;

public class MergeSort {

private static void mergeSort(int[] data, int start, int end) {

if (end > start) {

int pos = (start + end) / 2;

mergeSort(data, start, pos);

mergeSort(data, pos + 1, end);

merge(data, start, pos, end);

}

}

private static void merge(int[] data, int start, int pos, int end) {

int len1 = pos - start + 1;

int len2 = end - pos;

int A[] = new int[len1 + 1];

int B[] = new int[len2 + 1];

for (int i = 0; i < len1; i++) {

A[i] = data[i + start - 1];

}

A[len1] = Integer.MAX\_VALUE;

for (int i = 0; i < len2; i++) {

B[i] = data[i + pos];

}

B[len2] = Integer.MAX\_VALUE;

int m = 0, n = 0;

for (int i = start - 1; i < end; i++) {

if (A[m] > B[n]) {

data[i] = B[n];

n++;

} else {

data[i] = A[m];

m++;

}

}

}

private static void print(int[] data) {

for (int i = 0; i < data.length; i++) {

System.out.print(data[i] + " ");

}

}

public static void main(String args[]) {

int data[] = { 8, 16, 99, 732, 10, 1, 29, 66 };

print(data);

System.out.println();

mergeSort(data, 1, data.length);

print(data);

}

}